https://zhuanlan.zhihu.com/p/43789231

HTTPS（SSL/TLS）的加密机制虽然是大家都应了解的基本知识，但网上很多相关文章总会忽略一些内容，没有阐明完整的逻辑脉络，我学习它的时候也曾废了些功夫。

对称与非对称加密、数字签名、数字证书等，在学习过程中，除了了解"它是什么"，你是否有想过"为什么是它"？我认为有必要搞清楚后者，否则你可能只是单纯地记住了被灌输的知识，而未真正理解它。

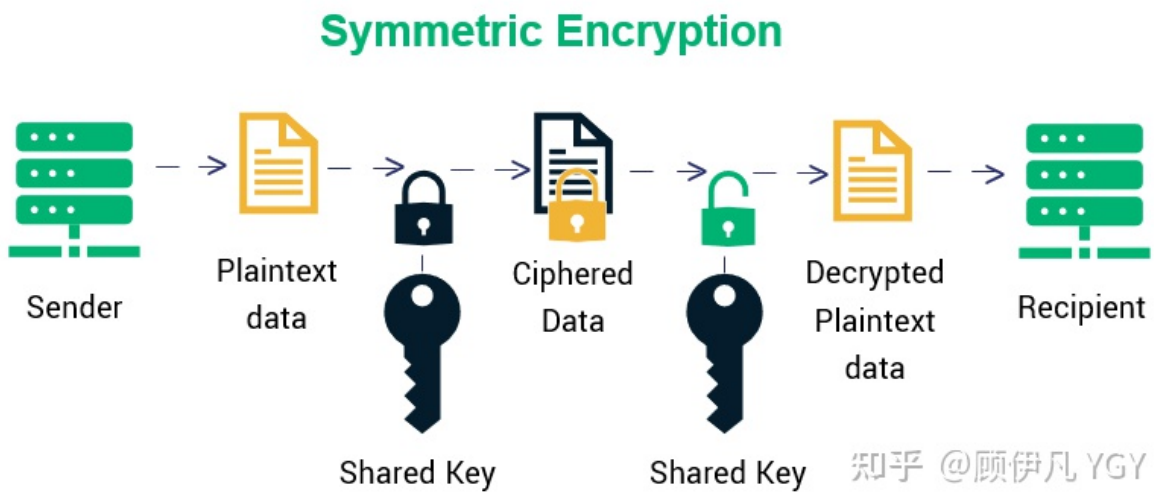
本文以问题的形式逐步展开，一步步解开HTTPS的面纱，希望能帮助你彻底搞懂HTTPS！

为什么需要加密？

因为http的内容是明文传输的，明文数据会经过中间代理服务器、路由器、wifi热点、通信服务运营商等多个物理节点，如果信息在传输过程中被劫持，传输的内容就完全暴露了。劫持者还可以篡改传输的信息且不被双方察觉，这就是**中间人攻击**。所以我们才需要对信息进行加密。最容易理解的就是`对称加密`。

什么是对称加密？

简单说就是有一个密钥，加密和解密用的是同一个秘钥。它可以加密一段信息，也可以对加密后的信息进行解密，和我们日常生活中用的钥匙作用差不多。



对称加密（https://sectigostore.com/blog/types-of-encryption-what-to-know-about-symmetric-vs-asymmetric-encryption/）

用对称加密可行吗？

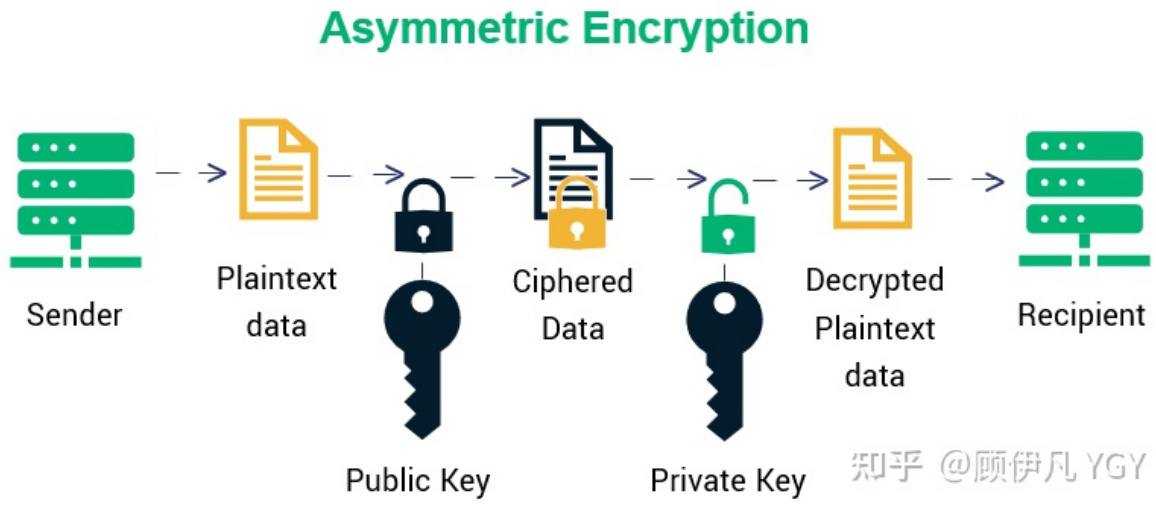
如果通信双方都各自持有同一个密钥，且没有别人知道，这两方的通信安全当然是可以被保证的（除非密钥被破解）。

然而最大的问题就是**这个密钥怎么让传输的双方知晓，同时不被别人知道。**如果由服务器生成一个密钥并传输给浏览器，那在这个传输过程中密钥被别人劫持到手了怎么办？之后他就能用密钥解开双方传输的任何内容了，所以这么做当然不行。

换种思路？试想一下，如果浏览器内部就预存了网站A的密钥，且可以确保除了浏览器和网站A，不会有任何外人知道该密钥，那理论上用对称加密是可以的，这样浏览器只要预存好世界上所有HTTPS网站的密钥就行了！这么做显然不现实。
怎么办？所以我们就需要`非对称加密`。

什么是非对称加密?

简单说就是有**两把密钥**，通常一把叫做公钥、一把叫私钥，用公钥加密的内容必须用私钥才能解开，同样，私钥加密的内容只有公钥能解开。



非对称加密（https://sectigostore.com/blog/types-of-encryption-what-to-know-about-symmetric-vs-asymmetric-encryption/）

用非对称加密可行吗?

鉴于非对称加密的机制，我们可能会有这种思路：服务器先把公钥以明文方式传输给浏览器，之后浏览器向服务器传数据前都先用这个公钥加密好再传，这条数据的安全似乎可以保障了！**因为只有服务器有相应的私钥能解开公钥加密的数据。**

然而反过来**由服务器到浏览器的这条路怎么保障安全?**如果服务器用它的私钥加密数据传给浏览器，那么浏览器用公钥可以解密它，而这个公钥是一开始通过明文传输给浏览器的，若这个公钥被中间人劫持到了，那他也能用该公钥解密服务器传来的信息了。所以**目前似乎只能保证由浏览器向服务器传输数据的安全性**（其实仍有漏洞，下文会说），那利用这点你能想到什么解决方案吗?

-- 给浏览器的公钥被中间替换了，那么整个链路都可以被篡改，双线都是不安全的。

改良的非对称加密方案，似乎可以?

我们已经理解通过一组公钥私钥，可以保证单个方向传输的安全性，那用两组公钥私钥，是否就能保证双向传输都安全了？请看下面的过程：

1. 某网站服务器拥有公钥A与对应的私钥A’；浏览器拥有公钥B与对应的私钥B’。
2. 浏览器把公钥B明文传输给服务器。
3. 服务器把公钥A明文给传输浏览器。
4. 之后浏览器向服务器传输的内容都用公钥A加密，服务器收到后用私钥A’解密。由于只有服务器拥有私钥A’，所以能保证这条数据的安全。
5. 同理，服务器向浏览器传输的内容都用公钥B加密，浏览器收到后用私钥B’解密。同上也可以保证这条数据的安全。

的确可以！抛开这里面仍有的漏洞不谈（下文会讲），HTTPS的加密却没使用这种方案，为什么？很重要的原因是非对称**加密算法非常耗时**，而对称加密快很多。那我们能不能运用非对称加密的特性解决前面提到的对称加密的漏洞?

-- 跟上面一样，如果被中间替换了。整个环节也是不安全的。都是伪造的

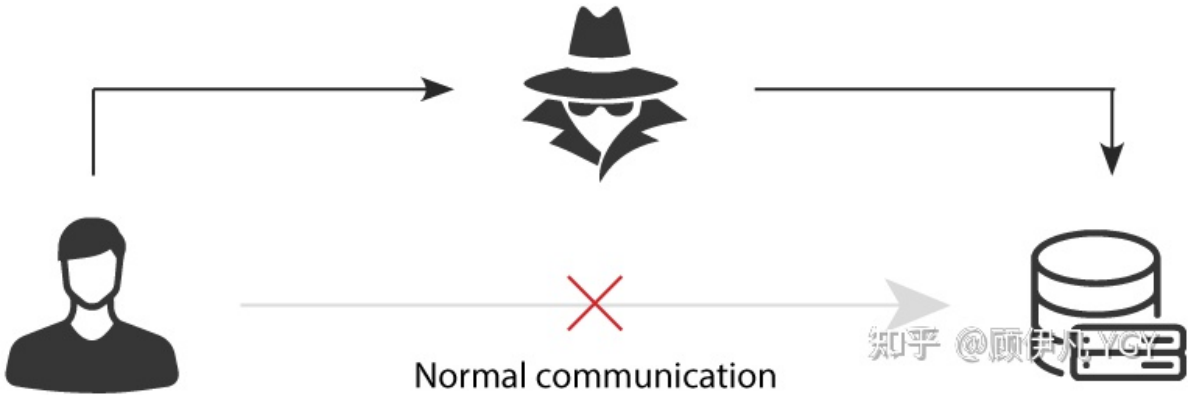
非对称加密+对称加密?

既然非对称加密耗时，那非对称加密+对称加密结合可以吗？而且得尽量减少非对称加密的次数。当然是可以的，且非对称加密、解密各只需用一次即可。
请看一下这个过程：

1. 某网站拥有用于非对称加密的公钥A、私钥A'。
2. 浏览器向网站服务器请求，服务器把公钥A明文给传输浏览器。
3. 浏览器随机生成一个用于对称加密的密钥X，用公钥A加密后传给服务器。
4. 服务器拿到后用私钥A'解密得到密钥X。
5. 这样双方就都拥有密钥X了，且别人无法知道它。之后双方所有数据都通过密钥X加密解密即可。

完美！HTTPS基本就是采用了这种方案。完美？还是有漏洞的。

中间人攻击



中间人攻击（https://blog.pradeo.com/man-in-the-middle-attack）

如果在数据传输过程中，中间人劫持到了数据，此时他的确无法得到浏览器生成的密钥X，这个密钥本身被公钥A加密了，只有服务器才有私钥A'解开它，然而中间人却完全不需要拿到私钥A'就能干坏事了。请看：

1. 某网站有用于非对称加密的公钥A、私钥A'。
2. 浏览器向网站服务器请求，服务器把公钥A明文给传输浏览器。
3. **中间人劫持到公钥A，保存下来，把数据包中的公钥A替换成自己伪造的公钥B（它当然也拥有公钥B对应的私钥B'）。**
4. 浏览器生成一个用于对称加密的密钥X，用**公钥B**（浏览器无法得知公钥被替换了）加密后传给服务器。
5. **中间人劫持后用私钥B'解密得到密钥X，再用公钥A加密后传给服务器。**
6. 服务器拿到后用私钥A'解密得到密钥X。

这样在双方都不会发现异常的情况下，中间人通过一套**"狸猫换太子"**的操作，掉包了服务器传来的公钥，进而得到了密钥X。**根本原因是浏览器无法确认收到的公钥是不是网站自己的，**因为公钥本身是明文传输的，难道还得对公钥的传输进行加密？这似乎变成鸡生蛋、蛋生鸡的问题了。解法是什么？

如何证明浏览器收到的公钥一定是该网站的公钥？

其实所有证明的源头都是一条或多条不证自明的"公理"（可以回想一下数学上公理），由它推导出一切。比如现实生活中，若想证明某身份证号一定是小明的，可以看他身份证，而身份证是由政府作证的，这里的"公理"就是"政府机构可信"，这也是社会正常运作的前提。

那能不能类似地有个机构充当互联网世界的"公理"呢？让它作为一切证明的源头，给网站颁发一个"身份证"？

它就是**CA机构**，它是如今互联网世界正常运作的前提，而CA机构颁发的"身份证"就是**数字证书。**

数字证书

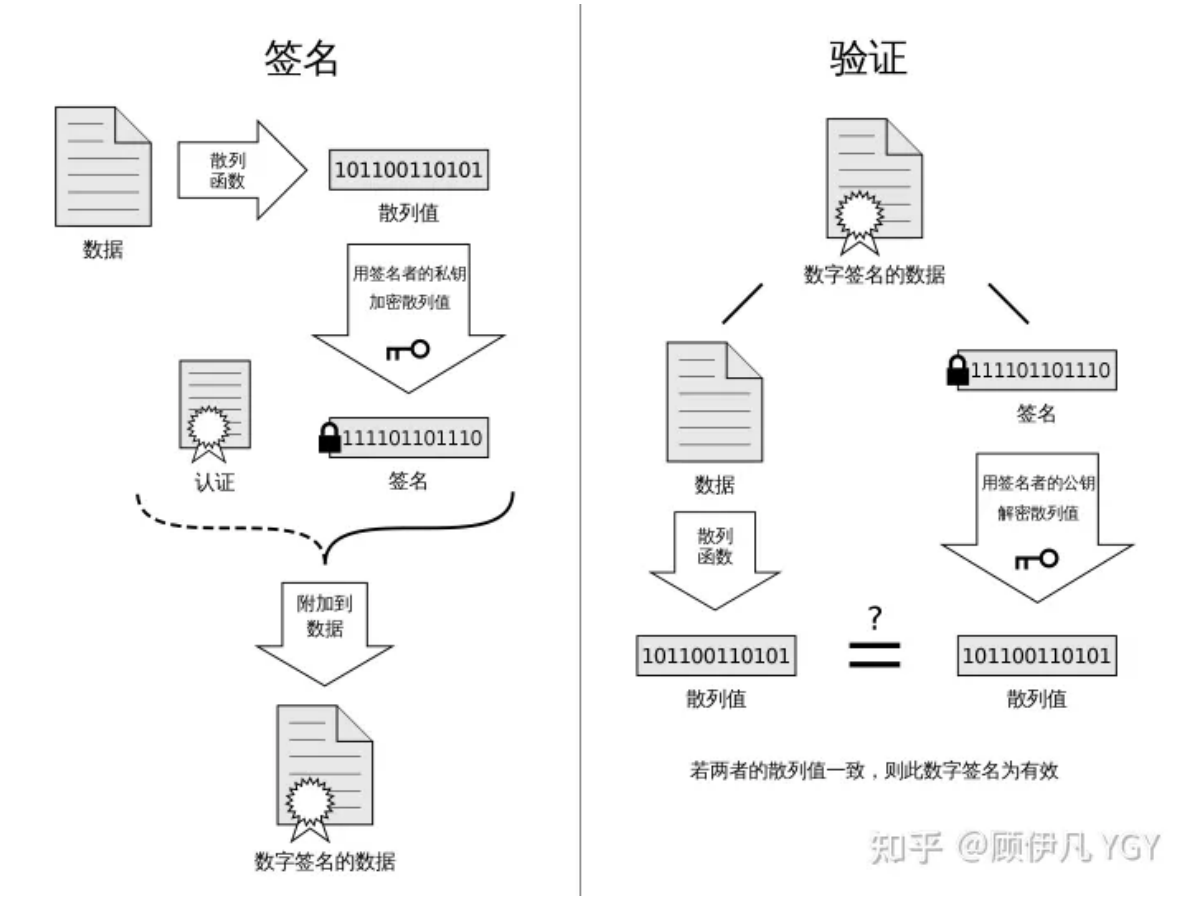
服务器在使用HTTPS前，需要向**CA机构**申领一份**数字证书**，认证机构在判明申请者的身份之后，会给申请者颁发证书。数字证书里含有证书持有者信息、公钥信息等。服务器把这个申请到的证书传输给浏览器，浏览器从证书里获取公钥就行了，证书就如身份证，证明"该公钥对应该网站"。而这里又有一个显而易见的问题，"**证书本身的传输过程中，如何防止被篡改**"？即如何证明证书本身的真实性？身份证运用了一些防伪技术，**而数字证书怎么防伪呢**？解决这个问题我们就接近胜利了！

如何放防止数字证书被篡改?

我们把证书原本的内容生成一份"签名"，比对证书内容和签名是否一致就能判别是否被篡改。这就是数字证书的"防伪技术"，这里的"签名"就叫"数字签名"：

数字签名

这部分内容建议看下图并结合后面的文字理解，图中左侧是数字签名的制作过程，右侧是验证过程：



数字签名的生成与验证（https://cheapsslsecurity.com/blog/digital-signature-vs-digital-certificate-the-difference-explained/）

数字签名的制作过程：

1. CA机构拥有非对称加密的私钥和公钥。
2. CA机构对证书明文数据T进行hash。
3. 对hash后的值用私钥加密，得到数字签名S。

明文和数字签名共同组成了数字证书，这样一份数字证书就可以颁发给网站了。
那浏览器拿到服务器传来的数字证书后，如何验证它是不是真的？（有没有被篡改、掉包）

浏览器验证过程：

1. 拿到证书，得到明文T，签名S。
2. 用CA机构的公钥对S解密（**由于是浏览器信任的机构，所以浏览器保有它的公钥。详情见下文**），得到S'。
3. 用证书里指明的hash算法对明文T进行hash得到T'。
4. 显然通过以上步骤，T'应当等于S'，除非明文或签名被篡改。所以此时比较S'是否等于T'，等于则表明证书可信。

为何么这样可以保证证书可信呢？我们来仔细想一下。

中间人有可能篡改该证书吗？

假设中间人篡改了证书的原文，由于他没有CA机构的私钥，所以无法得到此时加密后签名，无法相应地篡改签名。浏览器收到该证书后会发现原文和签名解密后的值不一致，则说明证书已被篡改，证书不可信，从而终止向服务器传输信息，防止信息泄露给中间人。

既然不可能篡改，那整个证书被掉包呢？

中间人有可能把证书掉包吗？

假设有另一个网站B也拿到了CA机构认证的证书，它想劫持网站A的信息。于是它成为中间人拦截到了A传给浏览器的证书，然后替换成自己的证书，传给浏览器，之后浏览器就会错误地拿到B的证书里的公钥了，这确实会导致上文"中间人攻击"那里提到的漏洞？

其实这并不会发生，因为证书里包含了网站A的信息，包括域名，浏览器把证书里的域名与自己请求的域名比对一下就知道有没有被掉包了。

为什么制作数字签名时需要hash一次？

我初识HTTPS的时候就有这个疑问，因为似乎那里的hash有点多余，把hash过程去掉也能保证证书没有被篡改。

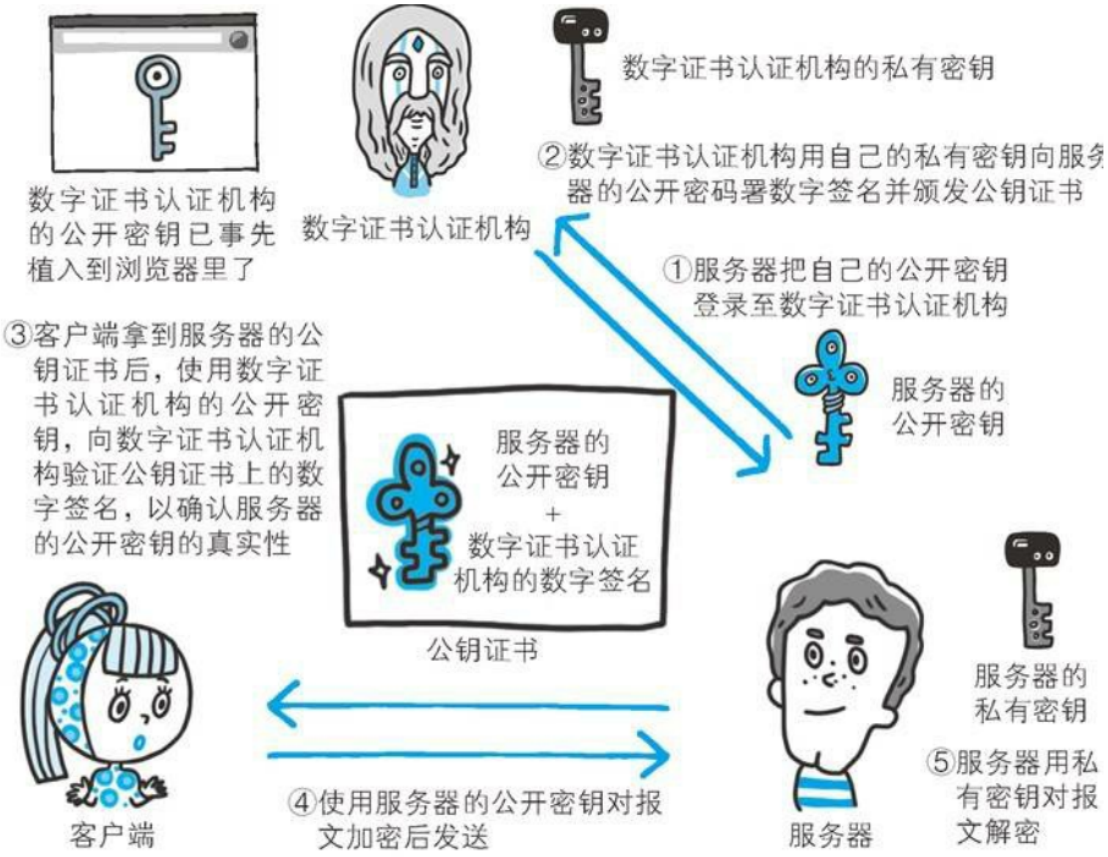
最显然的是性能问题，前面我们已经说了非对称加密效率较差，证书信息一般较长，比较耗时。而hash后得到的是固定长度的信息（比如用md5算法hash后可以得到固定的128位的值），这样加解密就快很多。

当然也有安全上的原因，这部分内容相对深一些，感兴趣的可以看这篇解答：
crypto.stackexchange.com/a/12780

怎么证明CA机构的公钥是可信的？-- 这才是所有的流程关键

你们可能会发现上文中说到CA机构的公钥，我几乎一笔带过，"浏览器保有它的公钥"，这是个什么保有法？怎么证明这个公钥是否可信？

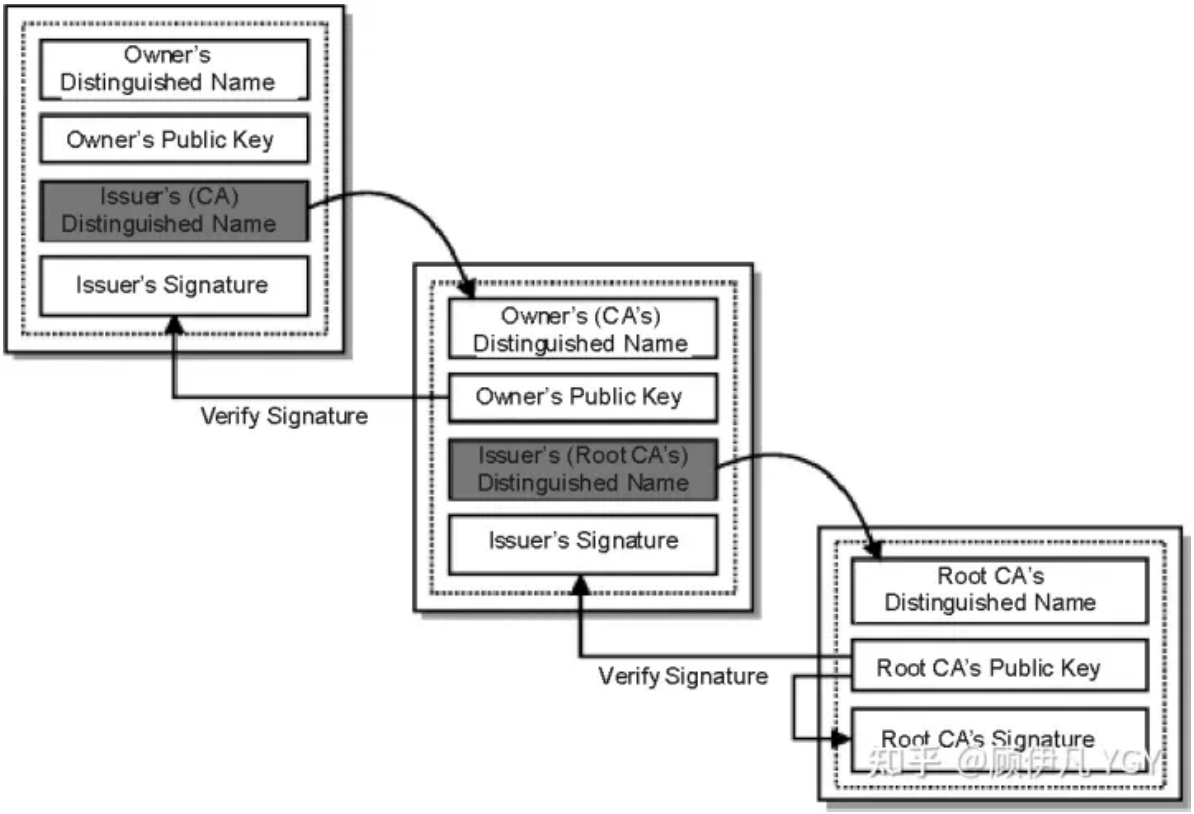
让我们回想一下数字证书到底是干啥的？没错，为了证明某公钥是可信的，即"该公钥是否对应该网站"，那CA机构的公钥是否也可以用数字证书来证明？没错，**操作系统、浏览器本身会预装一些它们信任的根证书，如果其中会有CA机构的根证书，这样就可以拿到它对应的可信公钥了。**



== 其实跟银行的U盾一个意思。所有的基础就是要保证公钥的安全传递

实际上证书之间的认证也可以不止一层，可以A信任B，B信任C，以此类推，我们把它叫做`信任链`或`数字证书链`。也就是一连串的数字证书，由根证书为起点，透过层层信任，使终端实体证书的持有者可以获得转授的信任，以证明身份。

另外，不知你们是否遇到过网站访问不了、提示需安装证书的情况？这里安装的就是根证书。说明浏览器不认给这个网站颁发证书的机构，那么你就得手动下载安装该机构的根证书（风险自己承担XD）。安装后，你就有了它的公钥，就可以用它验证服务器发来的证书是否可信了。



信任链（https://publib.boulder.ibm.com/tividd/td/TRM/GC32-1323-00/en_US/HTML/admin230.htm）

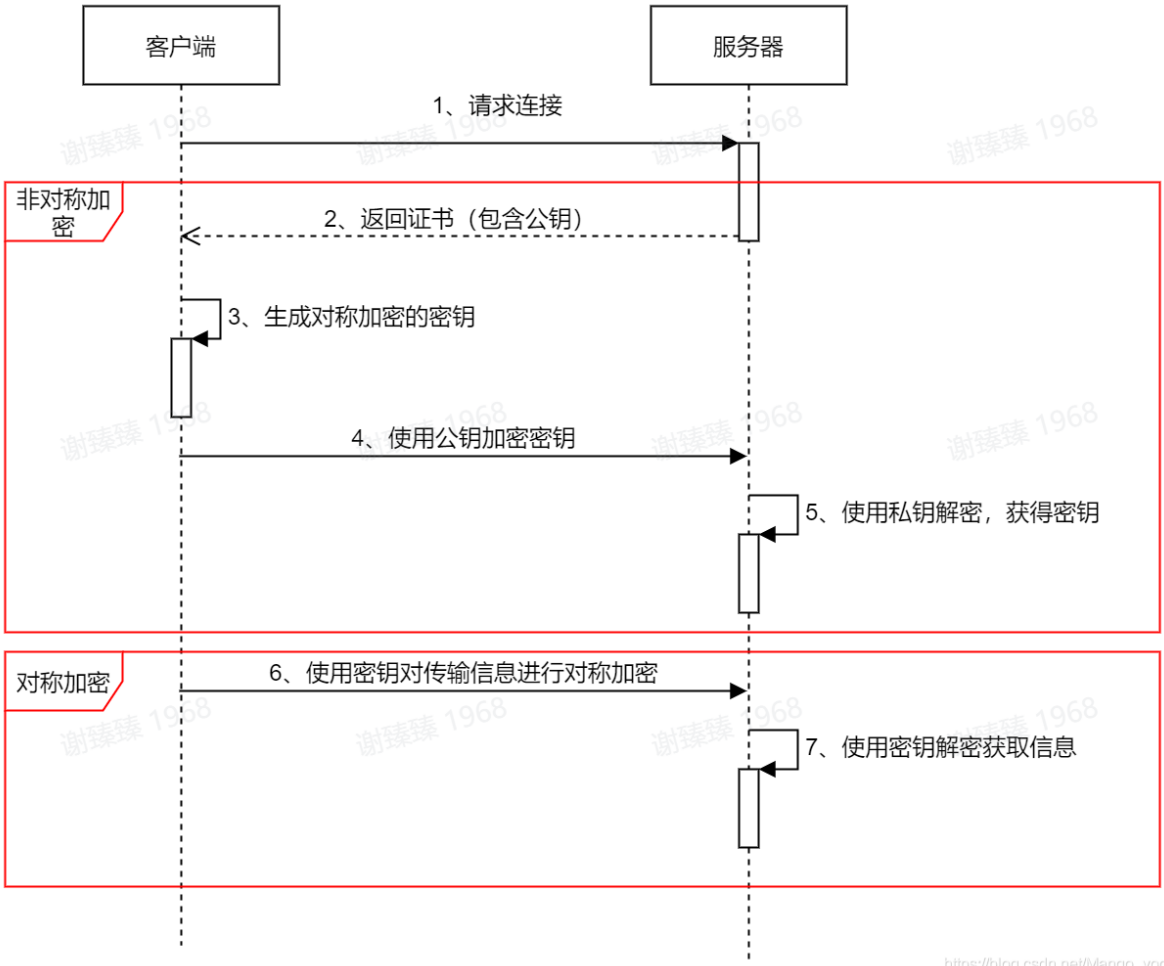
每次进行HTTPS请求时都必须在SSL/TLS层进行握手传输密钥吗?

这也是我当时的困惑之一，显然每次请求都经历一次密钥传输过程非常耗时，那怎么达到只传输一次呢?

服务器会为每个浏览器（或客户端软件）维护一个session ID，在TLS握手阶段传给浏览器，浏览器生成好密钥传给服务器后，服务器会把该密钥存到相应的session ID下，之后浏览器每次请求都会携带session ID，服务器会根据session ID找到相应的密钥并进行解密加密操作，这样就不必要每次重新制作、传输密钥了!

HTTPS采用混合加密机制

非对称加密速度慢，耗资源。所以https在交换密钥环节使用非对称密钥加密方式，之后的建立通信交换报文阶段则使用对称加密方式。

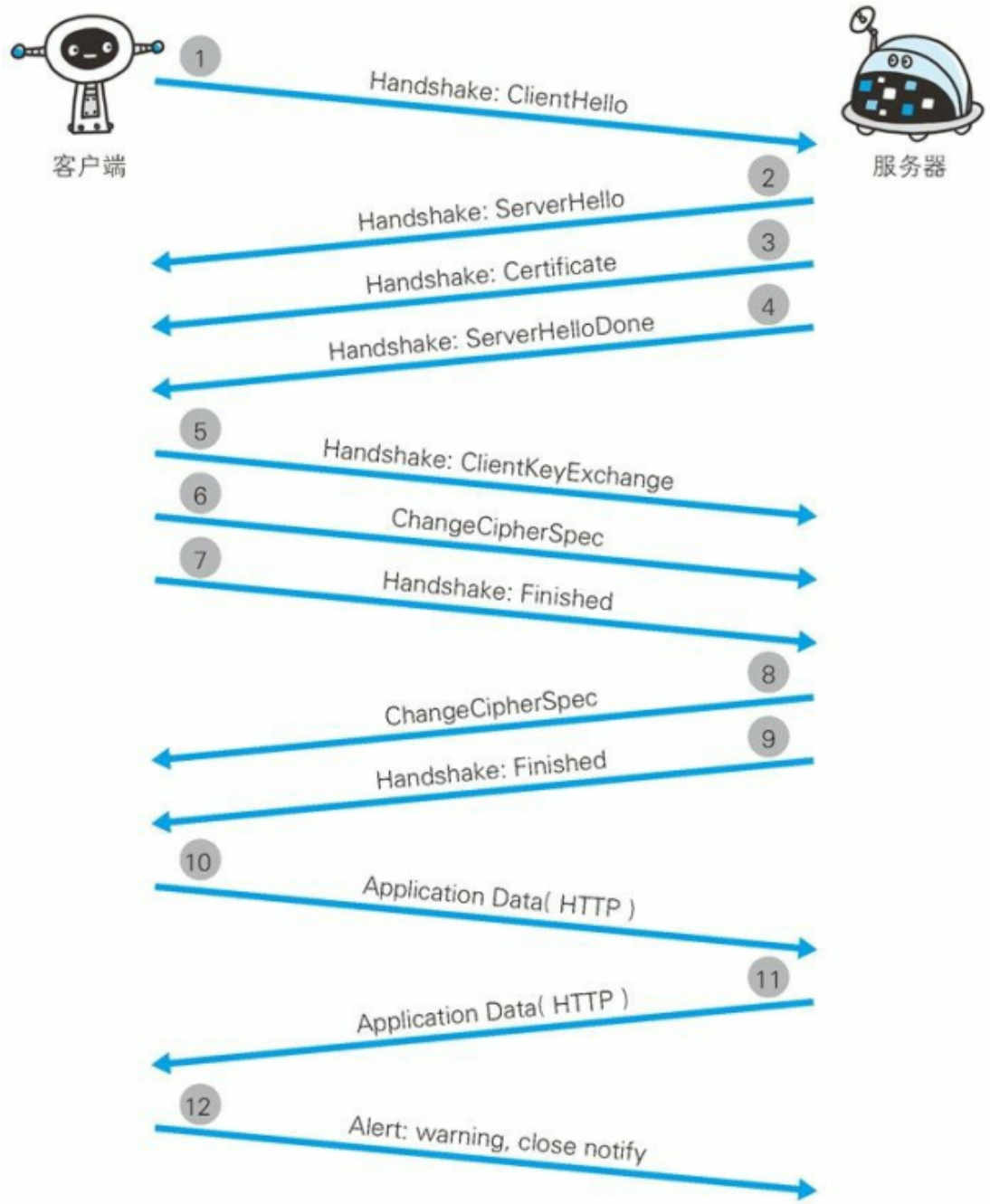


公开密钥加密处理起来比共享密钥加密方式更为复杂，因此若在通信时使用公开密钥加密方式，效率就很低

①使用公开密钥加密方式安全地交换在稍后的共享密钥加密中要使用的密钥

②确保交换的密钥是安全的前提下，使用共享密钥加密方式进行通信

…

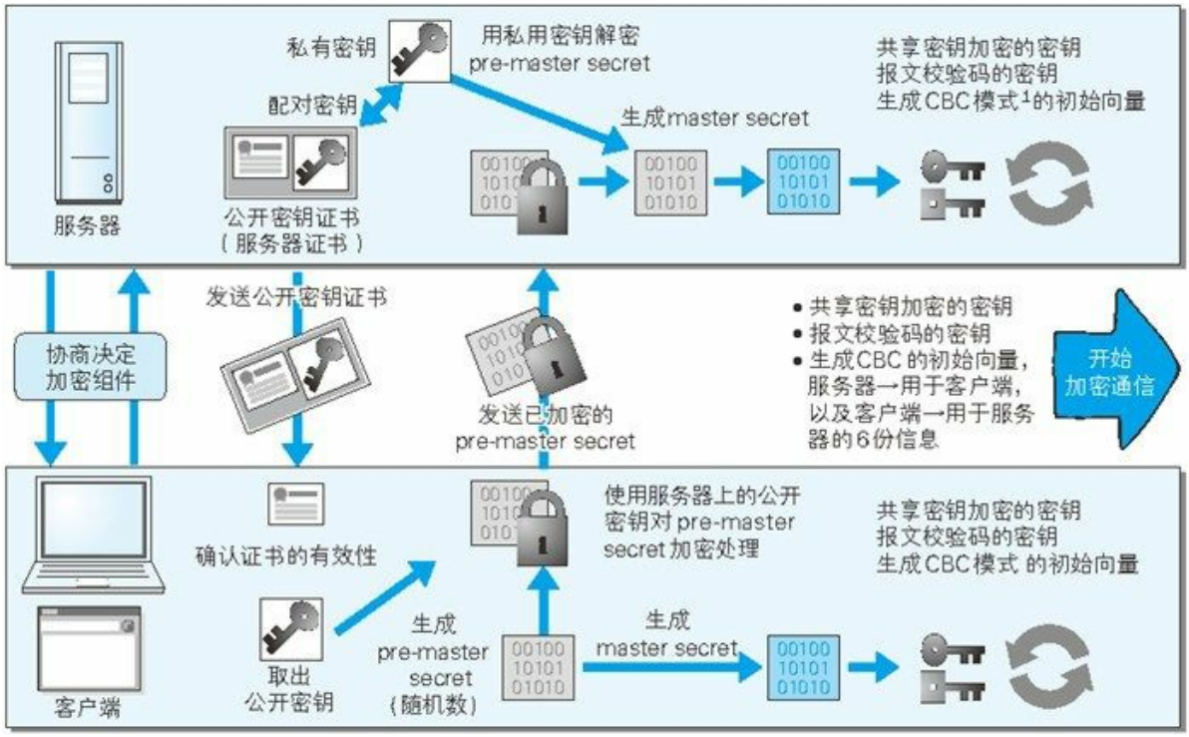


步骤 1： 客户端通过发送 Client Hello 报文开始 SSL通信。 报文中包含客户端支持的 SSL的指定版本、 加密组件（Cipher Suite） 列表（所使用的加密算法及密钥长度等） 。
步骤 2： 服务器可进行 SSL通信时， 会以 Server Hello 报文作为应答。 和客户端一样， 在报文中包含 SSL版本以及加密组件。 服务器的加密组件内容是从接收到的客户端加密组件内筛选出来的。
步骤 3： 之后服务器发送 Certificate（**证书**） 报文。 报文中包含公开密钥证书。

步骤 4： 最后服务器发送 Server Hello Done 报文通知客户端， 最初阶段的 SSL握手协商部分结束。

步骤 5： SSL第一次握手结束之后， 客户端以 Client Key Exchange（**客户端密钥交换**） 报文作为回应。 报文中包含通信加密中使用的一种被称为 Pre-master secret（**预主密钥**） 的随机密码串。 **该报文已用步骤 3 中的公开密钥进行加密。**
步骤 6： 接着客户端继续发送 Change Cipher Spec （**更改密码规范**）报文。 该报文会提示服务器， 在此报文之后的通信会采用 Pre-master secret 密钥加密。
步骤 7： 客户端发送 Finished 报文。 该报文包含连接至今全部报文的整体**校验值**。 这次握手协商是否能够成功， 要以服务器是否能够正确解密该报文作为判定标准。

步骤 8： 服务器同样发送 Change Cipher Spec 报文。
步骤 9： 服务器同样发送 Finished 报文。
步骤 10： 服务器和客户端的 Finished 报文交换完毕之后， SSL连接就算建立完成。 当然， 通信会受到 SSL的保护。 从此处开始进行应用层协议的通信， 即发送 HTTP 请求。
步骤 11： 应用层协议通信， 即发送 HTTP 响应。
步骤 12： 最后由客户端断开连接。 断开连接时， 发送 close_notify 报文。 上图做了一些省略， 这步之后再发送 TCP FIN 报文来关闭与 TCP的通信。
在以上流程中， 应**用层发送数据时会附加一种叫做 MAC（Message Authentication Code） 的报文摘要。** MAC 能够查知报文是否遭到篡改， 从而保护报文的完整性。
下面是对整个流程的图解。 图中说明了从仅使用服务器端的公开密钥证书（服务器证书） 建立 HTTPS 通信的整个过程。



CBC 模式（Cipher Block Chaining） 又名密码分组链接模式。 在此模式下， 将前一个明文块加密处理后和下一个明文块做 XOR 运算， 使之重叠， 然后再对运算结果做加密处理。 对第一个明文块做加密时， 要么使用前一段密文的最后一块，要么利用外部生成的初始向量（initial vector， IV） 。 ——译者注
SSL 和 TLS
HTTPS 使用 SSL（Secure Socket Layer） 和 TLS（Transport Layer Security） 这两个协议。SSL技术最初是由浏览器开发商网景通信公司率先倡导的， 开发过 SSL3.0 之前的版本。 目前主导权已转移到 IETF（Internet Engineering Task Force， Internet 工程任务组） 的手中。IETF 以 SSL3.0 为基准， 后又制定了 TLS1.0、TLS1.1 和TLS1.2。 **TSL是以 SSL为原型开发的协议，** 有时会统一称该协议为 SSL。 当前主流的版本是 SSL3.0 和 TLS1.0。
由于 SSL1.0 协议在设计之初被发现出了问题， 就没有实际投入使用。 SSL2.0 也被发现存在问题， 所以很多浏览器直接废除了该协议版本。
SSL 速度慢吗

HTTPS 也存在一些问题， 那就是当使用 SSL时， 它的处理速度会变慢。

SSL的慢分两种。 **一种是指通信慢。 另一种是指由于大量消耗CPU 及内存等资源， 导致处理速度变慢。**和使用 HTTP 相比， 网络负载可能会变慢 2 到 100 倍。 除去和TCP 连接、 发送 HTTP 请求 • 响应以外， 还必须进行 SSL通信，因此整体上处理通信量不可避免会增加。
另一点是 SSL必须进行加密处理。 在服务器和客户端都需要进行加密和解密的运算处理。 因此从结果上讲， 比起 HTTP 会更多地消耗服务器和客户端的硬件资源， 导致负载增强。
针对速度变慢这一问题， 并没有根本性的解决方案， 我们会使用SSL加速器这种（专用服务器） 硬件来改善该问

题。 该硬件为SSL通信专用硬件， 相对软件来讲， 能够提高数倍 SSL的计算速度。 仅在 SSL处理时发挥 SSL加速器的功效， 以分担负载

确认访问者身份

1，BASIC认证：基本认证

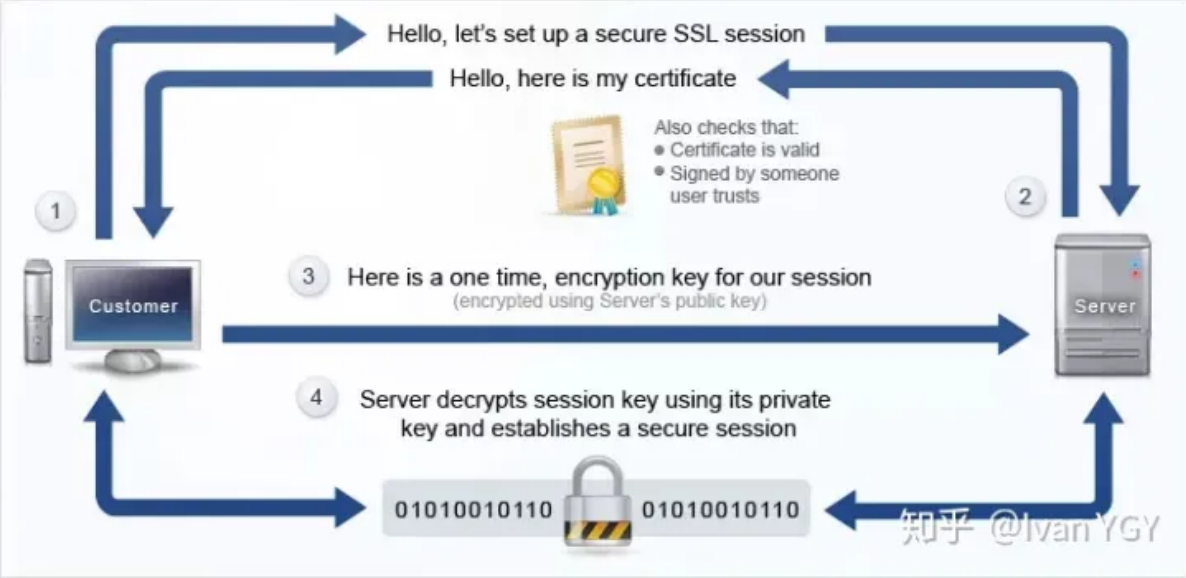
2，DIGEST认证：摘要认证

3，SSL客户端认证

4，FormBase认证（基于表单认证）

总结

可以看下这张图，梳理一下整个流程（SSL、TLS握手有一些区别，不同版本间也有区别，不过大致过程就是这样）：



（www.extremetech.com）

至此，我们已自上而下地打通了HTTPS加密的整体脉络以及核心知识点，不知你是否真正搞懂了HTTPS呢?